HP IO Accelerator Performance Tuning Guide

Abstract

This guide is designed to help verify and improve HP IO Accelerator performance.

Part Number: 652147-001
August 2011
Edition: 1

Contents

Introduction

About the Performance and Tuning Guide

Welcome to the Performance and Tuning Guide for the HP IO Accelerator. This guide is designed to help you achieve the following objectives:

- Verify IO Accelerator performance on Linux, including using sample benchmarks and solving common performance issues.
- Verify IO Accelerator performance on Windows® operating system, including using sample benchmarks and solving common performance issues.
- Understand and use effective benchmarking techniques to study and enhance IO Accelerator performance.
- Tune your system by describing common tuning techniques applicable to many systems and applications.
- Improve your application I/O capabilities by describing programming techniques to maximize performance.

System performance

Verifying Linux system performance

To verify Linux system performance with an IO Accelerator, HP recommends using the `fio` benchmark. This benchmark was developed by Jens Axboe, a Linux kernel developer. `Fio` is included in many distributions, or can be compiled from source. The latest source distribution (http://freshmeat.net/projects/fio) requires having the `libaio` development headers in place. For step-by-step instructions on compiling `fio` from source, see "Compiling the fio benchmark (on page 23)."

To test the raw performance of the IO Accelerator, HP recommends using raw block access. The best way to verify system performance is to run the `fio` tests in "Running fio tests (on page 6)."

△ **CAUTION:** The write tests destroy all data that currently resides on the IO Accelerator.

NOTE: For best performance on the IO Accelerator, use driver stack 2.1.0 or later.

Running fio tests

The sample jobs include the following four `fio` tests:

- One-card write bandwidth
- One-card read IOPS
- One-card read bandwidth
- One-card write IOPS

The following benchmarks are designed for maximally stressing the system and detecting performance issues, not to highlight IO Accelerator performance.

Benchmark tests

```
# Write Bandwidth test
$ fio --filename=/dev/fioa --direct=1 --rw=randwrite --bs=1m --size=5G
--numjobs=4 --runtime=10 --group_reporting --name=file1
# Read IOPS test
$ fio --filename=/dev/fioa --direct=1 --rw=randread --bs=4k --size=5G
--numjobs=64 --runtime=10 --group_reporting --name=file1
# Read Bandwidth test
$ fio --filename=/dev/fioa --direct=1 --rw=randread --bs=1m --size=5G
--numjobs=4 --runtime=10 --group_reporting --name=file1
# Write IOPS test
```

```
$ fio --filename=/dev/fioa --direct=1 --rw=randwrite --bs=4k --size=5G
--numjobs=64 --runtime=10 --group_reporting --name=file1
```

These tests are also available as `fio` job input files, which can be requested from HP support (http://www.hp.com/go/support).

The latest expected performance numbers for your card type can be found in the *HP PCIe IO Accelerator for ProLiant Servers Data Sheet* (*http://h20195.www2.hp.com/V2/GetPDF.aspx/4AA0-4235ENW.pdf*). Your results should exceed those on the data sheet. The sample IOPS test uses a 4K block size and 64 threads. The sample bandwidth test uses 1MB block size and four threads. For multi-card runs, IOPS are calculated by adding the per-card bandwidth/second together and dividing it by the block size. For sample output from each of the tests and how to validate that your system is performing properly, see "Write bandwidth test (on page 7)." The key data points in the output are highlighted.

Be sure to run the system-vetting tests in the order they appear in the following section. The initial write bandwidth test might need to be run twice with a short pause between the runs to setup the card for best performance.

Write bandwidth test

Read IOPS test on Linux:

```
file1: (g=0): rw=randread, bs=4K-4K/4K-4K, ioengine=sync, iodepth=1
...
file1: (g=0): rw=randread, bs=4K-4K/4K-4K, ioengine=sync, iodepth=1
Starting 64 processes
Jobs: 64 (f=64):
[rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr] [100.0%
done] [420482/          0 kb/s] [eta 00m:00s]
file1: (groupid=0, jobs=64): err= 0: pid=27861
     read : io=4, 058MiB, bw=414MiB/s, iops=104K, runt= 10036msec
           clat (usec) : min=44, max=36,940, avg=620.03, stdev=50.53
           bw (KiB/s) : min=5873, max=15664, per=1.55%, avg=6560.25, stdev=54.21
                 cpu : usr=0.50%, sys=4.01%, ctx=1163152, majf=0, minf=704
           IO depths : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%,
>64=0.0%
                 submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%,
>64=0.0%
           complete : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%,
>64=0.0%
           issued r/w: total=1038929/0, short=0/0
           lat (usec): 50=0.01%, 100=0.43%, 250=11.05%, 500=35.74%, 750=22.88%
           lat (usec): 1000=14.99%
           lat (msec): 2=14.88%, 4=0.02%, 10=0.01%, 50=0.01%


Run status group 0 (all jobs):
```

```
     READ: io=4,058MiB, aggrb= 414MiB/s, minb=414MiB/s, maxb=414MiB/s,
mint=10036msec, maxt=10036msec


Disk stats (read/write):
     fioa: ios=1038929/0, merge=0/0, ticks=389591/0, in_queue=61732674,
util=99.12%
```

The output below shows the test achieving 779 MiB/sec, with random reads done on 1MB blocks.

Read bandwidth test on Linux:

```
file1: (g=0): rw=randread, bs=1M-1M/1M-1M, ioengine=sync, iodepth=1
...
file1: (g=0): rw=randread, bs=1M-1M/1M-1M, ioengine=sync, iodepth=1
Starting 4 processes
Jobs: 4 (f=4): [rrrr] [100.0% done] [811597/          0 kb/s] [eta 00m:00s]
file1: (groupid=0, jobs=4): err= 0: pid=28781
     read : io=7, 614MiB, bw=779MiB/s, iops=____760____, runt= 1007msec
           clat (usec) : min=1,788, max=39,174, avg=5253.24, stdev=920.87
           bw (KiB/s) : min=157286, max=221412, per=25.16%, avg=200720.59,
stdev=5042.40
                 cpu : usr=0.03%, sys=2.79%, ctx=7880, majf=0, minf=1072
           IO depths : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%,
>64=0.0%
                 submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%,
>64=0.0%
           complete : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%,
>64=0.0%
           issued r/w: total=7614/0, short=0/0


           lat (msec): 2=0.39%, 4=15.34%, 10=83.18%, 20=0.99%, 50=0.11%


Run status group 0 (all jobs):
     READ: io=7,614MiB, aggrb= 779MiB/s, minb=779MiB/s, maxb=779MiB/s,
mint=1007msec, maxt=1007msec


Disk stats (read/write):
     fioa: ios=60912/0, merge=0/0, ticks=218927/0, in_queue=1667617,
util=98.84%
```

The test below achieved 106,000 IOPS, with random writes and a 4K block size.

Write IOPS test on Linux:

```
file1: (g=0): rw=randwrite, bs=4K-4K/4K-4K, ioengine=sync, iodepth=1
...
file1: (g=0): rw=randwrite, bs=4K-4K/4K-4K, ioengine=sync, iodepth=1
```

```
Starting 64 processes
Jobs: 64 (f=64):
[wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww] [100.0%
done] [          0/422981 kb/s] [eta 00m:00s]
file1: (groupid=0, jobs=64): err= 0: pid=28964
     write : io=4,138MiB, bw=424MiB/s, iops=106K, runt= 1001msec
           clat (usec): min=31, max=68,803, avg=589.54, stdev=67.72
           bw (KiB/s) : min= 0, max=17752, per=1.44%, avg=6231.17, stdev=182.09
     cpu              : usr=0.37%, sys=2.97%, ctx=10599830, majf=0, minf=576
     IO depths        : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%,
>64=0.0%
           submit     : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%,
>64=0.0%
           complete   : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%,
>64=0.0%
           issued r/w: total=0/1059304, short=0/0
           lat (usec): 50=0.21%, 100=0.67%, 250=1.81%, 500=2.79%, 750=94.36%
           lat (usec): 1000=0.14%
           lat (msec): 2=0.01%, 100=0.01%
Run status group 0 (all jobs):
     WRITE: io=4,138MiB, aggrb= 424MiB/s, minb=424MiB/s, maxb=424MiB/s,
mint=10001msec, maxt=10001msec


Disk stats (read/write):
     fioa: ios=0/1059304, merge=0/0, ticks=0/425609, in_queue=64011510,
util=99.50%
```

Verifying Windows system performance with Iometer

To verify IO Accelerator performance on a Windows® system, HP recommends the use of Iometer version 2006.07.27, which can be obtained from the Iometer website (http://www.iometer.org). This section describes sample tests and benchmarks for measuring IO Accelerator performance on Windows®.

For information on setting up Iometer, see "Using Iometer to verify IO Accelerator performance on Windows ("Using Iometer to verify IO Accelerator performance on Windows operating systems" on page 24)."

Iometer should be configured to use eight child threads, each with a queue of 64 outstanding requests. To achieve maximum performance, and for true verification results, the Iometer tests must be run against the raw block device, not against a filesystem.

The latest expected performance numbers for your card type can be found in the *HP PCIe IO Accelerator for ProLiant Servers Data Sheet (http://h20195.www2.hp.com/V2/GetPDF.aspx/4AA0-4235ENW.pdf)*. Your results should exceed those on the data sheet.

Debugging performance issues

Improperly configured benchmark

Issue

The most common issue in achieving performance with the IO Accelerator is the failure to properly set up the micro benchmark.

Solution

Be sure that you start with the benchmarks described in the previous sections to insure that the system is performing properly with a known benchmark.

Oversubscribed bus

Issue

It is possible to install multiple IO Accelerators and other PCIe peripherals in a way that causes unequal performance from each of the drives. In networking, this is frequently called over-subscription, and it is common in all but the latest generation of systems. For PCIe bus based high performance peripherals, having an over-subscribed topology can drastically affect performance, especially if the drives are set up for a RAID configuration.

An example of a balanced topology, which maximizes performance, is a system that has 16 PCI lanes connected from IO Accelerators to a switch chip, and 16 lanes from the switch chip to the root complex.

An example of an over-subscription condition, which decreases performance, is a system that has 12 PCI lanes connected from IO Accelerators to a switch chip, but only 8 lanes from the switch chip to the root complex.

Other PCIe devices, such as high performance network cards and graphic cards, can also create an over-subscription condition.

Solution

To verify that there are no bandwidth bottlenecks in the PCIe bus, run the `fio-pci-check` utility and look for errors.

> **NOTE:** The `fio-pci-check` utility is not fully functional on all operating systems.

Issue

A related common performance problem that is harder to diagnose is when the PCIe lanes are run off of the south bridge. Running off the south bridge is inherently slower than running off the north bridge. This can create an oversubscribed configuration that the `fio-pci-check` utility might not be able to diagnose.

Solution

Diagnosing this issue is beyond the scope of this document. For assistance in determining if your system suffers from this problem, run the `fio-bugreport` utility and the results of the system-vetting benchmarks.

Send the bundle to support@fusionio.com, and request assistance in debugging a performance issue. **[Reviewers: how do HP customers handle this?]**

Below is an example of a bandwidth error reported by fio-pci-check, caused by an over-subscription condition. Any line indicated with an asterisk indicates a possible problem detected by the `fio-pci-check` utility.

fio-pci-check

```
Root Bridge PCIe 3000 MB/sec
     Bridge 00:02.00 (01-05)
           *Needed 3000 MB/sec Avail 2000 MB/sec
                 Bridge 01:00.00 (02-04)
                       *Needed 3000 MB/sec Avail 2000 MB/sec
                             Bridge 02:00.00 (03-03)
                                   Needed 1000 MB/sec Avail 1000 MB/sec
                                         ioDrive 03:00.0 Firmware 14071
```

Handling PCIe errors

Issue

Data errors might be introduced when data is passed across the bus.

Solution

PCIe detects, and in many situations, corrects, data errors. These errors can also be detected by running the `fio-pci-check` utility.

> **NOTE:** The `fio-pci-check` utility is not fully functional on all operating systems.

If such errors are detected on your system:

1. Reseat all risers and IO Accelerators in the system.
2. Run some significant data to and from the IO Accelerator in question and check for errors again.
3. If the errors continue, the most common culprit is a riser card. Install a different riser card.
4. If errors persist, install a different motherboard.
5. If errors persist, install a new IO Accelerator.

The IO Accelerator causes PCI errors to show up faster than other devices do. Because of the high performance, the IO Accelerator is demanding on the PCI Express bus, showing errors in the system that other devices might not be able to uncover.

If multiple cards are installed in the system, use the `fio-status` and `fio-beacon` utilities to determine which of the drives is failing. The `fio-status` utility returns the serial number for the device, and the `fio-beacon` utility turns on the beacon LED for identification.

IMPORTANT: Some PCI Express chips do not properly report PCIe errors, or they might report errors when none exist. In most cases, this occurs on a bridge chip. This failure typically shows under the following conditions:

- Multiple rapid executions of the `fio-pci-check` utility were issued.
- No data is passing over the bus reporting errors.
- All drivers for attached peripherals are unloaded.

Below is an example of PCI Express errors captured on a system with an IO Accelerator.

NOTE: Windows® operating systems do not allow clearing of all errors, so only errors registered on the IO Accelerators can be considered.

```
Root Bridge PCIe 3000 MB/sec
     Bridge 00:02.00 (09-12)
           Needed 1000 MB/sec Avail 1000 MB/sec
                 Bridge 09:00.00 (0a-0f)
                       Needed 1000 MB/sec Avail 1000 MB/sec
                       * Fatal Error(s): Detected
`                      * Unsupported Type(s): Detected
                       Clearing Errors
                       Bridge 0a:00.00 (0b-0d)
                             Needed 1000 MB/sec Avail 1000 MB/sec
                                   ioDrive 0b:00.0 Firmware 14071
     Bridge 00:04.00 (13-15)
           Needed 1000 MB/sec Avail 1000 MB/sec
                 ioDrive 13:00.0 Firmware 14071
```

PCIe link width improperly negotiated

Issue

Currently shipping IO Accelerators are x4 PCI Express cards. If the system is having difficulties communicating with the IO Accelerator, the system might communicate with the IO Accelerator using only an x1 link, which has 1/4th the performance of an x4 link.

Solution

Use the `fio-pci-check` utility to check for this problem and to report any issues with link width.

NOTE: The `fio-pci-check` utility is not fully functional on all operating systems.

Below is an example of PCI link width errors captured on a system with an IO Accelerator.

```
Root Bridge PCIe 1750 MB/sec
     Bridge 00:01.00 (01-01)
           Needed 1000 MB/sec Avail 1000 MB/sec
     Current control settings: 0x000f
```

```
        Correctable Error Reporting: enabled
        Non-fatal Error Reporting: enabled
        Unsupported Request Reporting: enabled
    Current status: 0x0000
        Correctable Error(s): None
        Non-Fatal Error(s): None
        Fatal Error(s): None
        Unsupported Type(s): None
    Current link capabilities: 0x02014501
        Maximum link speed: 2.5 Gb/s
        Maximum link width: 16 lanes
    Current link capabilities: 0x00001041
        Link speed: 2.5 Gb/s
        Link width is 4 lanes
              ioDrive 01:00.0 Firmware 11791
```

CPU thermal throttling or auto-idling

Issue

Current-generation CPUs monitor themselves for thermal overheat. If the CPU is running on the edge of its manufacturer-specified thermal tolerance, the CPU will reduce its clock rate to keep from overheating. This reduces overall system performance, including IO Accelerator performance.

Some systems have a CPU idle detector installed as well. This decreases the clock rate of the CPU under low load to conserve power, but it also decreases overall system performance.

Solution

To detect a CPU running at a slower rate, inspect the contents of the `/proc/cpuinfo` file (in the Linux kernel) for discrepancies between the speed reported by the model and the current running speed. You must perform this inspection for each of the CPUs included in the listing.

Below is sample output of a system that has throttled from 2.66 GHz to 1 GHz.

```
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 15
model name : Intel (R) Xeon (R) CPU      5150 @ 2.66GHz
stepping : 5
cpu MHz : 1000.382
cache size : 4096 KB
--snip--
```

Benchmarking through a filesystem

Issue

Although using a filesystem is necessary for most storage deployments, it involves additional work to access the data stored on the IO Accelerator. These additional lookups decrease maximum system performance when compared to the benchmark results achieved by benchmarking directly on the block device.

Solution

When you are running micro-benchmarks to vet system performance, you should benchmark by accessing the block device directly. Otherwise, use any application-native filesystem implementation, possibly testing a handful where available. HP testing has shown XFS to be reasonably fast under most circumstances. For Linux, HP recommends using the XFS filesystem.

Slow performance using RAID5 on Linux

Issue

The native Linux implementation of RAID5 is verified to have performance issues with the IO Accelerator. The Linux RAID5 configuration is believed to use a single thread/single CPU to perform the needed parity calculations inherent in running a RAID5 system.

Solution

An alternate RAID stack might provide better performance. Additionally, IO Accelerators might be configured to operate in a RAID10 solution.

Using CP and other system utilities

Issue

Most traditional system utilities, such as `CP` and `rsync`, are built with slow legacy storage in mind. They do not achieve optimal performance from the IO Accelerator like well-tuned applications.

This is not to say that the IO Accelerator does not work well with standard utilities. It is still much faster than traditional storage using the same utilities, and additional performance benefits will be available in the future as these utilities are optimized for high-performance storage.

Solution

Avoid using traditional system utilities for general benchmarking purposes, as they are not a good representation of peak performance.

ext4 in Kernel.org 2.6.33 or earlier might silently corrupt data when discard (trim) is enabled

△ **CAUTION:** HP does not support the use of ext4 in Kernel.org 2.6.33 or earlier. Ext4 in Kernel.org 2.6.33 or earlier might silently corrupt data when discard is enabled.

The ext4 filesystem in the Kernel.org kernel 2.6.33 and earlier contains a bug where the data in a portion of a file might be improperly discarded (set to `all 0x00`) under some workloads. Use Version 2.6.34 or newer

to avoid this issue. For more information, see the patch (http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=b90f687018e6d6) and bug report (https://bugzilla.kernel.org/show_bug.cgi?id=15579).

The fix is included in RHEL6 pre-release kernel kernel-2.6.32-23.el6. The eventual release RHEL6 kernel is not affected by this issue.

Discard support was added to the kernel.org mainline ext4 in Version 2.6.28 and was enabled by default. For fear of damaging some devices, discard was set to default to disabled in Version 2.6.33-rc1 and was back ported to 2.6.31.8 and 2.6.32.1. For more information, see the kernel patch (http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=5328e635315734d).

General tuning techniques

Using direct I/O, unbuffered, or zero copy 10

Traditional I/O paths include the page cache, a DRAM cache of data stored on the disk. The IO Accelerator is fast enough that this and other traditional optimizations, such as I/O merging and reordering, are actually detrimental to performance. I/O merging and reordering are eliminated naturally by the IO Accelerator, but the page cache must be bypassed at the application level.

Direct I/O bypasses the page cache. This allows the same memory regions written by the application to be DMA-transferred or copied directly to the IO Accelerator rather than having to be copied into kernel-owned memory first, thus eliminating a copy.

Bypassing the page cache provides the following benefits:

- Less complex write path
- Lower overall CPU utilization
- Less memory bandwidth usage

In most cases, direct I/O is beneficial for IO Accelerator-based systems, but you should benchmark your application to be sure of this. Some applications that do not use AIO, threads, or multiple processes to create multiple outstanding requests might benefit from the page cache instead.

Many I/O-intensive applications have tunable parameters that control how they interact with the low-level IO subsystem, including turning on direct I/O.

For other applications, it is necessary for the application provider to enable direct I/O or to modify the source to enable direct I/O, and then recompile.

For a more in-depth look at how to write C code using Direct IO, see "Programming using Direct IO."

dd support

More recent versions of dd support the `oflag=direct` and `iflag=direc}}t` parameters. These enable direct I/O for either the file being written to or the file being read from, respectively. Use the `{{oflag=direct` parameter when writing to an IO Accelerator and the `iflag=direct` parameter when reading from an IO Accelerator.

ioZone Benchmark

ioZone supports the `-I` option to enable Direct I/O.

fio Benchmark

`Fio` uses the `direct=1` setting in the job file, or `--direct=1` on the command line to enable Direct I/O.

Multiple outstanding IOs

The IO Accelerator is more like a storage controller than a single disk. Like other storage controllers, it performs best when multiple requests are outstanding. Unlike other storage solutions that rely on legacy

protocols and have long pipelines, the IO Accelerator does not suffer from major latency increases as the number of outstanding I/Os increases.

The primary methods for generating outstanding I/Os are:

- Using multiple threads
- Using multiple processes
- Using AIO

For small-packet IOPS-geared applications, having multiple threads or outstanding AIO requests generally yields a significant performance improvement over a single thread. For larger block size bandwidth-oriented applications, having multiple outstanding I/Os is less important.

Pre-conditioning

Unlike traditional storage, the characteristics of writes issued to a solid state storage device can affect the performance of both future write and read operations. Some of the more interesting characteristics to consider are the size of individual writes (the block size or record size), the order in which writes are performed, and the block size used to read the data back. Providing the details for this is outside the scope of this document. The most common pre-conditioning issues are addressed here.

The `fio-format` command reinitializes the data on the IO Accelerator to an empty state. This eliminates all history of the data writes on the drive as well as removing all data. Deleting this history might initially cause higher performance results for both reads and writes. Ensure that the application and benchmark have had time to stabilize at a performance level.

Read performance can be artificially boosted when reads are performed from previously unwritten sectors.

After `fio-format` is complete, any sector that is read before data is written to it returns all binary zeros (0x0). It returns data at an accelerated rate when compared to data read from a sector that has previously had data written. This behavior is the same that filesystems and operating systems use to manipulate sparse files. The read performance achieved from these uninitialized sectors is not indicative of IO Accelerator real-world read performance and should be disregarded. The published numbers from HP disregard this performance acceleration.

To avoid measuring invalid read performance, ensure that you write data to each sector that will be used in benchmarking. In Linux, the entire device can be easily written to using the `dd` command:

CAUTION: The `dd` command destroys all data on the drive.

```
$ dd if=/dev/zero of=/dev/fioX bs=10M oflag=direct
```

Under Windows® operating systems, when a raw block test is being used, that same test can generally be used to write data to the device before testing. Testing that is run on top of a filesystem must first populate the data and cannot be affected by this artificial performance boost.

If an application writes in a smaller block size than it uses to read the data back, the read bandwidth might be constrained to the maximum bandwidth achievable at a block size equivalent to the original write block size.

For example, if an application performs random 512 byte writes and then reads the data back using 4 KiB, the performance might be limited to that of issuing 512 byte reads directly (the IO Accelerator is IOP limited rather than bandwidth limited.)

The most common ways to reset a device state are:

- Using the fio-format command to re-initialize the drive
- Performing large sequential writes to the drive

For more details on using the `fio-format` command, see the IO Accelerator User Guide for your operating system.

Increasing outstanding requests allowed by the kernel (Linux only)

This section applies to running the 2.0 driver series and later on Linux, with the default value `(3)` for the `use_workqueue driver` option.

The I/O schedulers limit the number of outstanding I/O requests issued to a block device to 128. This might limit achievable performance for small block random I/O. This limit can be adjusted dynamically. For example, to increase the number of allowed requests to 4096 per block device, use the following command:

```
$ echo 4096 > /sys/block/<fio name>/queue/nr_requests
```

where `<fio name>` is the block device name, such as `fioa`.

For some workloads, increasing this parameter might increase I/O latency.

Pre-allocating memory

The driver allocates memory as needed at runtime. If you need to keep memory allocations to a minimum, such as when using the IO Accelerator as a swap or page file, enable preallocation with the driver options described below.

You can further tune memory allocation for your IO Accelerator devices by using the following module parameters:

- preallocate_memory
- preallocate_mb
- expected_io_size

> **NOTE:** Improper use of the `preallocate_mb` or `expected_io_size` module parameters might cause the driver to have insufficient memory in the pre-allocate pool. If the driver needs more memory than is provided in the pre-allocate pool, it will fall back to non-pre-allocate behavior for the additional memory, allocating and freeing memory as it would under normal operation.

preallocate_memory

Description

Causes the driver to pre-allocate the RAM it needs for specified devices. A list of serial numbers is passed in so the driver can pre-allocate the maximum memory needed per device.

This module parameter does not normally need to be invoked if swap has been enabled. If the memory pre-allocation fails, the driver fails to load or enters minimal mode.

NOTE: HP recommends using this option only with a 4 KiB sector size.

Values	Description
<list of serial numbers>	Indicates devices that need memory pre-allocated. A null list disables the pre-allocation of memory.

preallocate_mb

Description

Specifies the number of megabytes of memory to pre-allocate. The sector size is set with the `fio-format` command-line utility using the `-b` option. Larger sector sizes result in less memory utilization.

Values	Description
0 (default)	Use the pre-allocate size calculator algorithm (do not specify the size of pre-allocated memory).
<size, MB>	Disable the pre-allocate size calculator algorithm and pre-allocate the amount of memory specified in MB.

expected_io_size

Description

This module parameter is useful when preallocate_memory is set through the Management Tool or a supplied list of serial numbers and preallocate_mb is left at the default (0).

It overrides the value provided to the pre-allocate size calculator algorithm. The default is built to handle worst-case memory utilization (which is the sector size of the device). For example, if your application writes I/O blocks larger than the sector size (such as when the drive is formatted for 512 byte sectors, but the application does 8 KiB I/O, then setting this parameter to 8192 when memory pre-allocation is enabled will significantly reduce the amount of pre-allocated memory.

Values	Description
0 (Default)	Don't pre-allocate memory
<size, MB>	Allocate the specified amount of memory, in megabytes.

Tuning techniques for writes

Increased steady-state write performance with fio-format

Under a sufficiently long sustained workload, write performance decreases. This performance drop, or steady state write performance, is common to all solid-state storage technologies. Even though most enterprise workloads induce steady state write behavior in IO Accelerator devices, they have high enough native performance that most applications see little to no performance penalty.

Most enterprise workloads trigger steady state write performance to some degree. The magnitude of the slowdown is closely tied to the nature of the workload. Non-application-based benchmarks (micro benchmarks) typically induce a larger performance hit than application-based performance tests. Actual application workloads are often even less impacted by steady state write.

For the most write-intensive applications, there are tuning options. The first line of tuning is down-formatting or under-formatting, in which a portion of the presented capacity of the device is returned to the device to be used for internal maintenance. Changing the presented capacity can be done using the `fio-format` utility or with the Management Tool. Repeated `fio-formats` are not required to get proper performance, unless the initial change in reserve size did not yield the needed steady-state write performance.

With a 2.x series driver and using a single IO Accelerator, a good starting point is to decrease the drive capacity by 10%. For example, if the drive is a 160 GB drive, you would `fio-format` it to 144GB. The maximum amount you can decrease the drive capacity by is 50%.

As the number of IO Accelerator devices increases, the percentage of down format required for a particular workload to achieve the target performance decreases.

CAUTION: This procedure destroys any existing data on your drives. If you already have data on a drive, be sure to back it up before proceeding.

Assuming the drive that you would like to format is `/dev/fct0` (use the `fio-status` utility to determine this), enter the following commands:

```
$ fio-detach /dev/fct0
$ fio-format -s 144G /dev/fct0
$ fio-attach /dev/fct0
```

The larger the reserve, the better the steady-state performance achieved.

Linux filesystem tuning

ext2-3-4 tuning

XFS is currently the recommended filesystem. It can achieve up to three times the performance of a tuned ext2/ext3 solution. At this time, there is no known additional tuning for running XFS in a single- or multi-IO Accelerator configuration.

Setting stride size and stripe width for ext2/3 (extN) when using RAID

The extN filesystem family has create-time options of stride and stripe width. Stride helps the filesystem ensure that critical metadata structures are spread across the disks in the RAID evenly, to keep any given disk from becoming a hotspot. Stripe width allows for filesystem optimization.

To calculate the correct values for stride size, take the chunk size of the RAID array and divide it by the block size of the filesystem. For example:

```
stride = (chunk size / filesystem block size)
```

Stripe width calculation requires the stride size and the number of data-bearing disks. The following table shows the calculation for number of data bearing disks per raid type.

```
dbd = # of data bearing disks

total_disks = # of active disks

mirrored_sets = The number of raid 1 mirrored sets that are used to form the
higher-level group
```

RAID level	Data bearing disks (dbd)
0 (Striping)	total_disks
1 (Mirroring)	1
5	total_disks - 1
6	total_disks - 2
10	mirrored_sets
50	mirrored_sets - 1

To calculate the stripe width, take the number of data bearing disks and multiply it by the stride.

```
stripe_width = dbd * stride
```

Sample Configuration

NOTE: The example below is not a recommended configuration, but it is a good demonstration of how to use the above equations.

Create a RAID50 with 10 IO Accelerators combined into five mirrored sets, (two mirrored IO Accelerators per set) and 256KB chunk size.

Create an ext3 filesystem with an 8K block size:

```
(stride) 32K = 256K / 8K
(dbd) 4 = 5 - 1
(stripe_width) 128 = 4 * 32K
```

This results in the following command:

```
$ mkfs.ext3 -b 8192 -E stride=32 -E stripe_width=128 /dev/md0
```

For more information on setting the stripe size, see the article "Optimizing the EXT3 file system on CentOS (http://wiki.centos.org/HowTos/Disk_Optimization)."

Using the IO Accelerator as swap space

NOTE: Adding multiple swap partitions usually slows down swap performance, even if they are set at the same priority. For best performance, only use one IO Accelerator device as swap space.

To safely use the IO Accelerator as swap space requires passing the `preallocate_memory` kernel module parameter. The recommended method for providing this parameter is to add the following line to the `/etc/modprobe.d/iomemory-vsl.conf` file:

```
options iomemory-vsl preallocate_memory=1072,4997,6710,10345
```

Where `1072`, `4997`, `6710`, and `10345` are serial numbers obtained from the `fio-status` utility.

A 4 KiB sector size format is required for swap. This reduces the driver memory footprint to reasonable levels. Use the Management Tool or the `fio-format` utility to format the IO Accelerator with 4 KiB sector sizes.

△ **CAUTION:** You must have 400 MiB of free RAM per 80GB of IO Accelerator capacity (formatted to 4 KiB block size) to enable the IO Accelerator with pre-allocation enabled for use as swap. Attaching an IO Accelerator with pre-allocation enabled without sufficient RAM may result in the loss of user processes and system stability.

NOTE: Be sure to provide the serial numbers for the IO Accelerators, not the adapter.

NOTE: The `preallocate_memory` module parameter is necessary to have the drive usable as swap space. See the *HP IO Accelerator User Guide for Linux* for more information on setting this parameter.

NOTE: The `preallocate_memory` parameter is recognized by the IO Accelerator driver at load time, but the requested memory is not actually allocated until the specified device is attached.

fio benchmark

Compiling the fio benchmark

The fio benchmarking utility is used to verify Linux system performance with an IO Accelerator. To compile the fio utility, get the latest version of fio source from the tarball link (http://freshmeat.net/projects/). For a tar balls of all fio releases, see the Index of /snaps (http://brick.kernel.dk/snaps).

1. Install the necessary standard dependencies, for example:

   ```
   gcc $ yum -y install gcc
   ```

2. Install the necessary `libaio` development headers, for example:

   ```
   $ yum -y install libaio-devel
   ```

3. Explode the fio tarball:

   ```
   $ tar xjvf fio-X.Y.Z.tar.bz2
   $ cd fio-X.Y.Z
   ```

4. Build fio for your system:

   ```
   $ make
   ```

When the rebuild completes successfully, a fio binary is placed in the fio directory.

Verifying IO Accelerator performance on Windows operating systems

Using Iometer to verify IO Accelerator performance on Windows operating systems

To set up an IO Accelerator to work with Iometer:

1. Ensure that you have the latest driver and firmware for the target IO Accelerator. For best results, be sure you are using a capable quad-core or higher CPU.
2. Load Iometer and add at least eight threads and 64 outstanding I/O's per target.
3. Select the drive to test for each thread.
4. Click the **Access Specifications** tab, and then select the test you want to run.
5. Modify the test configuration according to the test criteria. In this case, the configuration is set measure 4 KiB sequential throughput.

   NOTE: Make sure to match the Transfer Request Size values with the Align I/Os on values in the dialog box. Do not align the I/Os on Sector Boundaries, or the performance figures might be drastically lower.

6. Select the **Access Specification** for each worker thread.
7. Adjust the slider bar to set the **Update Frequency** to one second.
8. Click the **Save** button, and save the test configuration file.
9. Run the configured test to fill the drive with data and get results.

Programming using direct I/O

Using direct I/O on Linux

Under Linux, the best way to enable direct I/O is on a per-file basis. This is done by using the `O_DIRECT` flag to the open() system call. For example, in an application written in C, you might see a line similar to this:

```
fd = open(filename, O_WRONLY);
```

To make this file accessible through unbuffered or direct I/O, change the line to the following:

```
fd = open(filename, O_WRONLY | O_DIRECT );
```

The IO Accelerator requires that all I/O performed on a device using `O_DIRECT` must be 512-byte aligned and a multiple of 512 bytes in size. Buffers used to read data to and write data from must be page-size aligned.

```
open(), getpagesize(), and posix_memalign().
```

The following is a simple application that writes a pattern to the entire IO Accelerator in 1 MiB chunks, using `O_DIRECT`: Generally, performance can be slightly enhanced by using `pwrite` instead of `write`.

```
#define _XOPEN_SOURCE 600
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#define __USE_GNU
#include <fcntl.h>
#include <string.h>
#define FILENAME "/dev/fiob"
int main( int argc, char **argv)
{
void *buf;
int ret = 0;
int ps = getpagesize();
unsigned long long int bytes_written = 0;
int fd;
if( (ret = posix_memalign(&buf, ps, ps*256)) ) {
perror("Memalign failed");
exit(ret);
}
```

```
memset(buf, 0xaa, ps*256);
if( (fd = open(FILENAME, O_WRONLY | O_DIRECT) ) < 0 ) {
perror("Open failed");
exit(ret);
}
bytes_written = 0;
while( (ret = pwrite(fd, buf, ps*256, bytes_written)) == ps*256) {
bytes_written += ret;
}
printf("Wrote %lld GB\n", bytes_written/1000/1000/1000);
close(fd);
free(buf);
}
```

Using direct I/O on Windows

Direct I/O on Windows operating systems is set up through the `CreateFile()` call, using the `FILE_FLAG_NO_BUFFERING` flag. For more information, see the "CreateFile Function (http://msdn.microsoft.com/en-us/library/aa363858(VS.85).aspx)" topic on MSDN.

When using direct IO, HP recommends that you keep read, writes, and buffers used for read/write sector size aligned. For a discussion of alignment requirements, see the "File Buffering (http://msdn.microsoft.com/en-us/library/cc644950(VS.85).aspx)" topic on MSDN. The HP IO Accelerator sector size defaults to 512 bytes.

C++ code sample

The following code sample writes to the block device using `O_DIRECT`. The code sample has a class that encapsulates the operating system file functions that support `O_DIRECT`.

```
#include <unistd.h>
#include <iostream>
#include <errno.h>
#include <fcntl.h>
using namespace std;
// Short demonstration of a container class that writes using O_DIRECT.
// Compiled using GNU C++.
#define FILENAME "/dev/fioc"
class DirectFile
{
public:
DirectFile() {};
~DirectFile() { ::close(fd); };
```

```
int open(char* filename);
int write(char* buf, int size_t);
int close();
size_t gbytesWritten() { return bytes_written / 1000 / 1000 / 1000; }
private:
int fd;
size_t bytes_written;
};
int DirectFile::open(char* filename) {
int ret = 0;
fd = ::open(filename, O_DIRECT | O_WRONLY, S_IRUSR | S_IWUSR | S_IRGRP);
if( fd == -1 ) {
ret = -1;
}
bytes_written = 0;
return ret;
}
int DirectFile::write(char* buf, int bufsize) {
int written = pwrite(fd, buf, bufsize, bytes_written);
if( written >= 0 )
bytes_written += written;
return written;
}
int main( int argc, char **argv) {
char* buf;
int ret = 0;
enum {
numpages = 256,
};
int pagesize = getpagesize();
int bufsize = pagesize * numpages;
if( ret = posix_memalign((void**)&buf, pagesize, bufsize) ) {
cerr << "Memalign failed" << endl;
exit(ret);
}
memset(buf, 0xaa, bufsize);
DirectFile file;
if( file.open(FILENAME) ) {
cerr << "Open of " << FILENAME << " failed" << endl;
```

```
exit(ret);
}
do {
ret = file.write(buf, bufsize);
if( ret < 0 ) {
cerr << endl << "Error writing bytes to " << FILENAME <<
". written=" << file.gbytesWritten() << "GBytes";
cerr << ", errno=" << errno << " " << strerror(errno) << endl;
} else {
cout << ".";
}
} while( ret > 0 );
if( ret >= 0 )
cout << endl << "Wrote " << file.gbytesWritten() << "GBytes" <<
endl;
free(buf);
return ret;
}
```

Windows driver affinity

Setting Windows driver affinity

On a multiprocessor system, the operating system routes an I/O request through as efficient a path as its programming permits. Often this path is not the optimal performance path, primarily due to system architecture. A user who is aware of the particular hardware layout of a system can maximize driver performance by specifying the routing of its I/O. The HP IO Accelerator Windows® driver provides a mechanism to specify an affinity of its I/O to a particular processor or set of processors. This helps the operating system route the requests through a more efficient path.

Many multiprocessor systems employ a uniform method of routing request threads. These threads receive a relatively equal amount of processor time from any processor assigned by the operating system. More recently, some systems have been developed with the NUMA architecture which couples a subset of total physical memory with a node containing one or more processors. One advantage to the NUMA architecture is the improvement in throughput of operations that can be handled using a particular processor and its locally associated memory. The disadvantage to NUMA becomes apparent when the operation must take place between a particular processor and memory that is physically associated with a processor on a different node.

To overcome this disadvantage, the operating system permits applications to programatically specify the affinity of an operation, interrupt, or a thread with a particular processor or set of processors. For the Windows Server® 2008 operating system, the IO Accelerator driver has been updated to accept user-specified values that represent processor masks. These masks enable interrupts and worker threads to be associated with the processors bitmapped by the masks.

To implement one of these values:

1. Open the Windows® Registry editor.
2. In the Windows® Registry, locate the following key:

   `HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services/fiodrive/Parameters/`
3. Create a Windows® Registry tag of type `REG_DWORD`, and then name it `SetWorkerAffinity?`

The `?` is replaced by a decimal value equal to the bus number of the IO Accelerator card for which the affinity is being set. This bus number is the same decimal value that appears in the `fct?` display after an IO Accelerator utility such as `fio-status` has run.

To make the tag effective in the driver, reboot the system or enable the driver, assuming defaults for other `fiodrive` tags.

Example

For example, if your system employs a NUMA architecture with four modular nodes, then each of which contains 2 GB of local memory and a dual-core CPU. From specifications, the machine contains eight PCIe buses, numbered 2, 11, 36, 37, 82, 86, 169, and 170 (decimal). You further determine that the buses are directly associated with specific nodes, as follows:

- Buses 2 and 11 : Node 0 (processors 0 and 1)

- Buses 36 and 37 : Node 1 (processors 2 and 3)
- Buses 82 and 86 : Node 2 (processors 4 and 5)
- Buses 169 and 170 : Node 3 (processors 6 and 7)

Because the Windows Server® operating system designates each processor according to its node numbering, it assigns the two processors of node 0 as processor 0 and processor 1. These are represented internally by a bitmask whose offsets correspond to the processor numbers. So, the processors on node 0 are represented by a 32-bit mask of b00000000000000000000000000000011, or more conveniently, 0x00000003.

To set the affinity for the Windows® IO Accelerator driver:

> **CAUTION:** Implementing these settings requires a firm knowledge of the physical layout of the system. Incorrect settings can degrade system performance.

1. Open the Windows® Registry Editor.
2. Navigate to the **Parameters** folder.
3. Create the `SetWorkerAffinity2` tag of type `REG_DWORD`.
4. Set the value to `0x03`.
5. Continue creating the remaining tags with their accompanying mask values as follows:

```
SetWorkerAffinity2 : 0x03
SetWorkerAffinity11 : 0x03
SetWorkerAffinity36 : 0x0C
SetWorkerAffinity37 : 0x0C
SetWorkerAffinity82 : 0x30
SetWorkerAffinity86 : 0x30
SetWorkerAffinity169 : 0xC0
SetWorkerAffinity170 : 0xC0
```

6. Either reboot (recommended) or disable the server, and then enable each instance of the driver.

Acronyms and abbreviations

AIO

asynchronous input/output

CPU

central processing unit

DMA

direct memory access

DRAM

dynamic random access memory

I/O

input/output

IOPS

input/output operations per second

LBA

logical block addressing

NUMA

Non-Uniform Memory Architecture

PCIe

peripheral component interconnect express

QDR

quad data rate

SSS

solid state storage

Index

Z